Testing Finite State Machines Presenting Stochastic Time and Timeouts*

Dept. Sistemas Informáticos y Programación
Universidad Complutense de Madrid, 28040 Madrid, Spain
mgmerayo@fdi.ucm.es,{mn,isrodrig}@sip.ucm.es

Abstract. In this paper we define a formal framework to test implementations that can be represented by the class of finite state machines introduced in [10]. First, we introduce an appropriate notion of test. Next, we provide an algorithm to derive test suites from specifications such that the constructed test suites are sound and complete with respect to two of the conformance relations introduced in [10]. In fact, the current paper together with [10] constitute a complete formal theory to specify and test the class of systems covered by the before mentioned stochastic finite state machines.

1 Introduction

The scale and heterogeneity of current systems make impossible for developers to have an overall view of them. Thus, it is difficult to foresee those errors that are either critical or more probable. In this line, *formal testing techniques* [8,14,3,15] allow to test the correctness of a system with respect to a specification. Formal testing originally targeted the functional behavior of systems, such as determining whether the tested system can, on the one hand, perform certain actions and, on the other hand, does not perform some non-expected ones. While the relevant aspects of some systems only concern *what* they do, in some other systems it is equally relevant *how* they do what they do. Thus, after the initial consolidation stage, formal testing techniques started also to deal with *non-functional* properties. In fact, there are already several proposals for timed testing (e.g. [9,4,16,5,11,12,7,6,2,13]). In these papers, with the only exception of [12], time is considered to be *deterministic*, that is, time requirements follow the form "after/before t time units..." In fact, in most of the cases time is introduced by means of clocks following [1]. Even though the inclusion of time allows to give a more precise description of the system to be implemented, there are frequent situations that cannot be accurately described by using this notion of deterministic time. For example, we may desire to specify a system where a message is expected to be received with probability $\frac{1}{2}$ in the interval $(0, 1]$, with probability $\frac{1}{4}$ in $(1, 2]$, and so on.

* Research partially supported by the Spanish MEC project WEST/FAST (TIN2006-15578-C02-01) and the Marie Curie project TAROT (MRTN-CT-2003-505121).

K. Wolter (Ed.): EPEW 2007, LNCS 4748, pp. 97–111, 2007.

In order to use a formal technique, we need that the systems under study can be expressed in terms of a formal language. A suitable representation of the temporal behavior is critical for constructing useful models of real-time systems. A language to represent these systems should enable the definition of temporal conditions that may direct the system behavior, as well as the time consumed by the execution of tasks. In this line, the time consumed during the execution of a system falls into one of the following categories:

(a) The system consumes time while it performs its operations. This time may depend on the values of certain parameters of the system, such as the available resources.
(b) The time passes while the system waits for a reaction from the environment. In particular, the system can change its internal state if an interaction is not received before a certain amount of time.

A language focussing on temporal issues should allow the specifier to define how the system behavior is affected by both kinds of temporal aspects. Even though there exists a myriad of timed extensions of classical frameworks, most of them specialize only in one of the previous variants: Time is either associated with actions or associated with delays/timeouts. In this paper we use the formalism introduced in [10] that allows to specify in a natural way both time aspects. In our framework, timeouts are specified by using fix amounts of time. In contrast, the duration of actions will be given by *random variables*. That is, we will have expressions such as "with probability p the action o will be performed before t units of time". We will consider a suitable extension of finite state machines where (stochastic) time information will be included. Intuitively, we will consider that the time consumed between the input is applied and the output is received is given by a random variable ξ. An appropriate notation for stochastic transitions could be $s \xrightarrow{i/o}_{\xi} s'$, meaning that "if the machine is in state s and receives an input i then it will produce the output o before time t with probability $P(\xi \leq t)$ and it will change its state to s'". The definition of conformance testing relations is more difficult than usually. In particular, even in the absence of non-determinism, the same sequence of actions may take different time values to be performed in different runs of the system. While the definition of the new language is not difficult, mixing these temporal requirements strongly complicates the posterior theoretical analysis.

As we have already indicated, this paper represents a continuation of the work initiated in [10]. In that paper we proposed several *stochastic-temporal conformance relations*: An implementation is correct with respect to a specification if it does not show any behavior that is forbidden by the specification, where both the functional behavior and the temporal behavior are considered (and, implicitly, how they affect each other). In this paper we introduce a notion of test and how to test implementations that can be represented by using our notion of finite state machine. In addition, we provide an algorithm that derives test suites from specifications. The main result of our paper indicates that these test suites have the same distinguishing power as the two most interesting conformance relations

presented in [10] in the sense that an implementation successfully passes a test suite iff it is conforming to the specification.

The rest of the paper is structured as follows. In the next two sections we remind our notion of stochastic finite state machine and the two most interesting implementation relations introduced in [10]. In Section 4 we formally define a notion of test, as well as the application of tests to implementations and two notions of successfully passing a test suite. In Section 5 we present an algorithm to derive test suites and show that the derived test suites appropriately capture the relations introduced in Section 3. Finally, in Section 6 we present our conclusions.

2 SFSM: A Stochastic Extension of the FSM Model

In this section we introduce our notion of finite state machines with stochastic time. We use random variables to model the (stochastic) time output actions take to be executed. Thus, we need to introduce some basic concepts on random variables. We will consider that the sample space, that is, the domain of random variables, is a set of numeric time values `Time`. Since this is a *generic* time domain, the specifier can choose whether the system will use a discrete/continuous time domain. We simply assume that $0 \in$ `Time`. Regarding passing of time, we will also consider that machines can evolve by raising *timeouts*. Intuitively, if after a given time, depending on the current state, we do not receive any input action then the machine will change its current state.

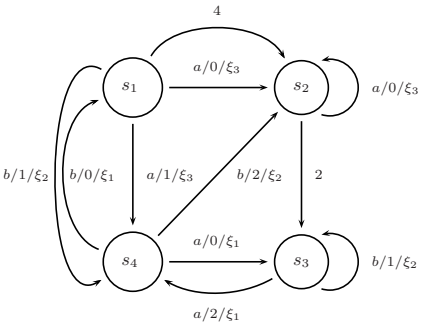
During the rest of the paper we will use the following notation. Tuples of elements $(e_1, e_2 \ldots, e_n)$ will be denoted by $\bar{e}$. â denotes an interval of elements $[a_1, a_2)$, with $a_1, a_2 \in$ `Time` and $a_1 < a_2$. We will use the projection function π_i such that given a tuple $\bar{t} = (t_1, \ldots, t_n)$, for all $1 \leq i \leq n$ we have $\pi_i(\bar{t}) = t_i$. Let $\bar{t} = (t_1, \ldots, t_n)$ and $\bar{t'} = (t'_1, \ldots, t'_n)$. We denote by $\sum \bar{t}$ the addition of all the elements belonging to the tuple $\bar{t}$, that is, $\sum_{j=1}^{n} t_j$. The number of elements of the tuple will be represented by $|\bar{t}|$. Finally, if $\bar{t} = (t_1 \ldots t_n)$, $\bar{p} = (\hat{t_1} \ldots \hat{t_n})$ and for all $1 \leq j \leq n$ we have $t_j \in \hat{t_j}$, we write $\bar{t} \in \bar{p}$.

Definition 1. We denote by $\mathcal{V}$ the set of random variables ($\xi, \psi, \ldots$ range over $\mathcal{V}$). Let ξ be a random variable. We define its *probability distribution function* as the function $F_\xi :$ `Time` $\longrightarrow [0, 1]$ such that $F_\xi(x) = P(\xi \leq x)$, where $P(\xi \leq x)$ is the probability that ξ assumes values less than or equal to x.

Given two random variables ξ and ψ we consider that $\xi + \psi$ denotes a random variable distributed as the addition of the two random variables ξ and ψ.

We will use the delimiters $\{\!|$ and $|\!\}$ to denote multisets. Given a set E, we denote by $\wp(E)$ the multisets of elements belonging to E. Given the multiset H over E, for all $r \in E$ we have that $H(r)$ denotes the *multiplicity* of r in H. Given two multisets H_1 and H_2 over E, $H_1 \uplus H_2$ denotes the union of H_1 and H_2, and it is formally defined as $(H_1 \uplus H_2)(r) = H_1(r) + H_2(r)$ for all $r \in E$.

We will call *sample* to any multiset of elements belonging to `Time`. Let ξ be a random variable and J be a sample. We denote by $\gamma(\xi, J)$ the *confidence* of ξ on J. □



$$F_{\xi_1}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ \frac{x}{5} & \text{if } 0 < x < 5 \\ 1 & \text{if } x \geq 5 \end{cases}$$

$$F_{\xi_2}(x) = \begin{cases} 0 & \text{if } x < 4 \\ 1 & \text{if } x \geq 4 \end{cases}$$

$$F_{\xi_3}(x) = \begin{cases} 1 - e^{-2 \cdot x} & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Fig. 1. Example of Stochastic Finite State Machine

In our setting, samples will be associated with the time values that implementations take to perform sequences of actions. We have that $\gamma(\xi, J)$ takes values in the interval $[0, 1]$. Intuitively, bigger values of $\gamma(\xi, J)$ indicate that the observed sample J is more likely to be produced by the random variable ξ. That is, this function decides how *similar* the probability distribution function generated by J and the one corresponding to the random variable ξ are.

In the appendix of [10] we show one of the possibilities to formally define the notion of confidence by means of a hypothesis contrast.

Definition 2. A *Stochastic Finite State Machine*, in short `SFSM`, is a tuple $M = (S, I, O, \delta, TO, s_{in})$ where S is the set of states, with $s_{in} \in S$ being the *initial state*, I and O denote the sets of input and output actions, respectively, δ is the set of transitions, and $TO : S \longrightarrow S \times (\texttt{Time} \cup \{\infty\})$ is the *timeout function*. Each transition belonging to δ is a tuple (s, i, o, ξ, s') where $s, s' \in S$ are the initial and final states, $i \in I$ and $o \in O$ are the input and output actions, and $\xi \in \mathcal{V}$ is the random variable defining the time associated with the transition.

Let $M = (S, I, O, \delta, TO, s_{in})$ be a `SFSM`. We say that M is *input-enabled* if for all state $s \in S$ and input $i \in I$ there exist s', o, ξ, such that $(s, i, o, \xi, s') \in \delta$. We say that M is *deterministically observable* if for all s, i, o there do not exist two different transitions $(s, i, o, \xi_1, s_1), (s, i, o, \xi_2, s_2) \in \delta$. □

Intuitively, a transition (s, i, o, ξ, s') indicates that if the machine is in state s and receives the input i then the machine emits the output o before time t with probability $F_\xi(t)$ and the machine changes its current state to s'.

For each state $s \in S$, the application of the timeout function $TO(s)$ returns a pair (s', t) indicating the time that the machine can remain at the state s waiting for an input action and the state to which the machine evolves if no input is received on time. We indicate the absence of a timeout in a given state by setting the corresponding time value to ∞. In addition, we assume that $TO(s) = (s', t)$ implies $s \neq s'$, that is, timeouts always produce a change of the state. In fact, let

us note that a definition such as $TO(s) = (s, t)$ is equivalent to set the timeout for the state s to infinite.

Example 1. Let us consider the machine depicted in Figure 1 in which the initial state is s_1. Each transition has an associated random variable. In the following we explain how these random variables are distributed. We consider that ξ_1 is *uniformly distributed* in the interval $[0, 5]$. Uniform distributions assign equal probability to all the times in the interval. The random variable ξ_2 follows a Dirac distribution in 4. The idea is that the corresponding delay will be equal to 4 time units. Finally, ξ_3 is *exponentially* distributed with parameter 2. Let us consider the transition $(s_4, (b, 0, \xi_1), s_1)$. Intuitively, if the machine is in state s_4 and it receives the input b then it will produce the output 0 after a time given by ξ_1. For example, we know that this time will be less than 1 time unit with probability $\frac{1}{5}$, it will be less than 3 time units with probability $\frac{3}{5}$, and so on. Finally, once 5 time units have passed we know that the output 0 has been performed (that is, we have probability 1). Regarding the timeout function we have $TO(s_1) = (s_2, 4)$. In this case, if the machine is in state s_1 and no input is received before 4 units of time then the state is changed to s_2.

Definition 3. Let $M = (S, I, O, \delta, TO, s_{in})$ be a `SFSM`. We say that a tuple $(s_0, s, i/o, \hat{t}, \xi)$ is a *step* for the state s_0 of M if there exist k states $s_1, \ldots, s_k \in S$, with $k \geq 0$, such that $\hat{t} = \left[\sum_{j=0}^{k-1} \pi_2(TO(s_j)), \sum_{j=0}^{k} \pi_2(TO(s_j))\right)$ and there exists a transition $(s_k, i, o, \xi, s) \in \delta$.

We say that $(\hat{t}_1/i_1/\xi_1/o_1, \ldots, \hat{t}_r/i_r/\xi_r/o_r)$ is a *stochastic evolution* of M if there exist r steps of M $(s_{in}, s_1, i_1/o_1, \hat{t}_1, \xi_1), \ldots, (s_{r-1}, s_r, i_r/o_r, \hat{t}_r, \xi_r)$ for the states $s_{in} \ldots s_{r-1}$, respectively. We denote by $\texttt{SEvol}(M)$ the set of stochastic evolutions of M. In addition, we say that $(\hat{t}_1/i_1/o_1, \ldots, \hat{t}_r/i_r/o_r)$ is a *functional evolution* of M. We denote by $\texttt{FEvol}(M)$ the set of functional evolutions of M. We will use the shortenings $(\sigma, \bar{p})$ and $(\sigma, \bar{p}, \bar{\xi})$ to denote a functional and a stochastic evolution, respectively, where $\sigma = (i_1/o_1 \ldots i_r/o_r)$, $\bar{p} = (\hat{t_1} \ldots \hat{t_r})$ and $\bar{\xi} = (\xi_1 \ldots \xi_r)$. □

Intuitively, a step is a sequence of transitions that contains an action transition preceded by zero or more timeouts. The interval $\hat{t}$ indicates the time values where the transition could be performed. In particular, if the sequence of timeouts is empty then we have the interval $\hat{t} = [0, TO(s_0))$. An evolution is a sequence of inputs/outputs corresponding to the transitions of a chain of steps, where the first one begins with the initial state of the machine. In addition, stochastic evolutions also include time information which inform us about possible timeouts (indicated by the intervals $\hat{t}_j$) and random variables associated to the execution of each output after receiving each input in each step of the evolution. In the following definition we introduce the concept of *instanced evolution*. Intuitively, instanced evolutions are constructed from evolutions by instantiating to a concrete value each timeout, given by an interval, of the evolution.

Definition 4. Let $M = (S, I, O, \delta, TO, s_{in})$ be a `SFSM` and let us consider a *stochastic evolution* $e = (\hat{t}_1/i_1/\xi_1/o_1, \ldots, \hat{t}_r/i_r/\xi_r/o_r)$. We say that the tuple

$(t_1/i_1/\xi_1/o_1, \ldots, t_r/i_r/\xi_r/o_r)$ is an *instanced stochastic evolution of* e if for all $1 \leq j \leq r$ we have $t_j \in \hat{t}_j$. Besides, we say that the tuple $(t_1/i_1/o_1, \ldots, t_r/i_r/o_r)$ is an *instanced functional evolution* of e.

We denote by $\texttt{InsSEvol}(M)$ the set of instanced stochastic evolutions of M and by $\texttt{InsFEvol}(M)$ the set of instanced functional evolutions of M. □

Example 2. Let us consider the SFSM depicted in Figure 1. Next, we give some of the *steps* that the machine can generate. For example, $(s_1, s_2, a/0, [0, 4), \xi_3)$ represents the transition from the state s_1 to the state s_2 when no timeouts precede it. The input a can be accepted before 4 time units pass (this is indicated by the interval $[0, 4)$). In addition, the output 0 takes t time units to be performed with probability $F_{\xi_3}(t)$. The second one, $(s_2, s_3, b/1, [2, \infty), \xi_2)$, is built from the timeout transition associated to the state s_2 and the transition outgoing the state s_3 to the state s_3. This step represents that if after 2 time units no input is received, the timeout transition associated with the state s_2 will be triggered and the state will change to s_3. After this, the machine can accept the input b. So, during the time interval $[2, \infty)$, if the machine receives an input b it will emit an output 1 and the machine remains at state s_3.

Now, we present an example of a stochastic evolution built from these steps and assuming that s_1 is the initial state: $([0, 4)/a/\xi_3/0, [2, \infty)/b/\xi_2/1)$. □

3 Implementation Relations

In this section we remind two of the implementation relations introduced in [10]. First, we give an implementation relation to deal with functional aspects. It follows the pattern borrowed from $\texttt{conf}_{nt}$ [11]: An implementation I *conforms* to a specification S if for all possible evolution of S the outputs that the implementation I may perform after a given input are a subset of those for the specification. In addition we require that the implementation always complies with the timeouts established by the specification. Besides the non-stochastic conformance of the implementation, we require other additional conditions, related to stochastic time, to hold.

We consider that specifications and implementations are given by means of SFSMs. We will consider that both of them are deterministically observable. Besides, we assume that input actions are always enabled in any state of the implementation, that is, implementations are input-enabled according to Definition 2. This is a usual condition to assure that the implementation will react (somehow) to any input appearing in the specification. First, we introduce the implementation relation $\texttt{conf}_f$, where only functional aspects of the system (i.e., which outputs are allowed/forbidden and how timeouts are defined) are considered while the performance of the system (i.e., how fast outputs are executed) is ignored. Let us note that the time spent by a system waiting for the environment to react has the capability of affecting the set of available outputs of the system. This is because this time may trigger a change of the state. So, a relation focusing on functional aspects must explicitly take into account the maximal time the system may stay in each state. This time is given by the *timeout* of each state.

Definition 5. Let S and I be SFSMs. We say that I *functionally conforms* to S, denoted by I $\mathtt{conf}_f$ S, if for each functional evolution $e \in \mathtt{FEvol}(S)$, with $e = (\hat{t}_1/i_1/o_1, \ldots, \hat{t}_r/i_r/o_r)$ and $r \geq 1$, we have that for all $t_1 \in \hat{t}_1, \ldots, t_r \in \hat{t}_r$ and o'_r, $e' = (t_1/i_1/o_1, \ldots, t_r/i_r/o'_r) \in \mathtt{InsFEvol}(I)$ implies $e' \in \mathtt{InsFEvol}(S)$. □

Intuitively, the idea underlying the definition of the functional conformance relation I $\mathtt{conf}_f$ S is that the implementation I does not *invent* anything for those sequences of inputs that are *specified* in the specification S. Let us note that if the specification has also the property of input-enabled then we may remove the condition "for each functional evolution $e \in \mathtt{FEvol}(S)$, with $e = (\hat{t}_{t1}/i_1/o_1, \ldots, \hat{t}_{tr}/i_r/o_r)$ and $r \geq 1$".

In addition to requiring this notion of *functional* conformance, we have to ask for some conditions on delays. A first approach would be to require that the random variables associated with evolutions of the implementation are identically distributed as the ones corresponding to the specification. However, the fact that we assume a black-box testing framework disallows us to check whether these random variables are indeed identically distributed. Thus, we have to give more *realistic* implementation relations based on finite sets of observations. Next, we present two implementation relations that are less *accurate* but that are *checkable*. These relations take into account the observations that we may get from the implementation. We will collect a sample of time values and we will *compare* this sample with the random variables appearing in the specification. By comparison we mean that we will apply a contrast to decide, with a certain confidence, whether the sample could be generated by the corresponding random variable.

Definition 6. Let I be a SFSM. We say that $(\sigma, \bar{t}, \bar{t}')$, with $\sigma = i_1/o_1, \ldots, i_n/o_n$, $\bar{t} = (t_1 \ldots t_n)$, and $\bar{t}' = (t'_1 \ldots t'_n)$, is an *observed time execution* of I, or simply *time execution*, if the observation of I shows that for all $1 \leq j \leq n$ we have that the time elapsed between the acceptance of the input i_j and the observation of the output o_j is t'_j units of time, being the input i_j accepted t_j units of time after the last output was observed.

Let $\Phi = \{(\sigma_1, \bar{t}_1), \ldots, (\sigma_m, \bar{t}_m)\}$ and let $H = \{\!\{(\sigma'_1, \bar{t}_{d1}, \bar{t}_{o1}), \ldots, (\sigma'_n, \bar{t}_{dn}, \bar{t}_{on})\}\!\}$ be a multiset of timed executions. We say that $\mathtt{Sampling}^k_{(H,\Phi)} : \Phi \longrightarrow \wp(\mathtt{Time})$ is a *k-sampling application* of H for Φ if $\mathtt{Sampling}^k_{(H,\Phi)}(\sigma, \bar{t}) = \{\!\{\pi_k(\bar{t}_o) \mid (\sigma, \bar{t}, \bar{t}_o) \in H \wedge |\sigma| \geq k\}\!\}$, for all $(\sigma, \bar{t}) \in \Phi$. We say that $\mathtt{Sampling}_{(H,\Phi)} : \Phi \longrightarrow \wp(\mathtt{Time})$ is a *sampling application* of H for Φ if $\mathtt{Sampling}_{(H,\Phi)}(\sigma, \bar{t})) = \{\!\{\sum \bar{t}_o \mid (\sigma, \bar{t}, \bar{t}_o) \in H\}\!\}$, for all $(\sigma, \bar{t}) \in \Phi$. □

Regarding the definition of k-sampling applications, we just associate with each subtrace of length k the observed time of each transition of the execution at length k. In the definition of sampling applications, we assign to each trace the total observed time corresponding to the whole execution.

Definition 7. Let I and S be SFSMs, H be a multiset of timed executions of I, $0 \leq \alpha \leq 1$, $\Phi = \{(\sigma, \bar{t}) \mid \exists\, \bar{t}_o : (\sigma, \bar{t}, \bar{t}_o) \in H\} \cap \mathtt{InsFEvol}(S)$, and let us consider $\mathtt{Sampling}_{(H,\Phi)}$ and $\mathtt{Sampling}^k_{(H,\Phi)}$, for all $1 \leq k \leq max\{|\sigma| \mid (\sigma, \bar{t}) \in \Phi\}$.

We say that I $(\alpha, H)-$*weak stochastically conforms* to S, and we denote it by I $\mathtt{confs}_w^{(\alpha,H)}$ S, if I $\mathtt{conf}_f$ S and for all $(\sigma, \bar{t}) \in \Phi$ we have

$$(\sigma, \bar{t}, \bar{\xi}) \in \mathtt{InsSEvol}(S) \Longrightarrow \gamma\left(\sum \bar{\xi}, \mathtt{Sampling}_{(H,\Phi)}(\sigma, \bar{t})\right) > \alpha$$

We say that I $(\alpha, H)-$*strong stochastically conforms* to S, and we denote it by I $\mathtt{confs}_s^{(\alpha,H)}$ S, if I $\mathtt{conf}_f$ S and for all $(\sigma, \bar{t}) \in \Phi$ we have

$$(\sigma, \bar{t}, \bar{\xi}) \in \mathtt{InsSEvol}(S) \Longrightarrow \forall\ 1 \leq j \leq |\sigma| : \gamma(\pi_j(\bar{\xi}), \mathtt{Sampling}^j_{(H,\Phi)}(\sigma, \bar{t})) > \alpha$$

□

The idea underlying the new relations is that the implementation must conform to the specification in the usual way (that is, I $\mathtt{conf}_f$ S). Besides, for all observation of the implementation that can be performed by the specification, the observed execution time values *fit* the random variable indicated by the specification. This notion of *fitting* is given by the function γ that it is formally defined in the appendix of [10]. While the *weak* notion only compares the total time, the *strong* notion checks that the time values are appropiate for each performed output.

4 Tests Cases for Stochastic Systems

We consider that tests represent sequences of inputs applied to an IUT. Once an output is received, the tester checks whether it belongs to the set of expected ones or not. In the latter case, a fail signal is produced. In the former case, either a pass signal is emitted (indicating successful termination) or the testing process continues by applying another input. If we are testing an implementation with input and output sets I and O, respectively, tests are deterministic acyclic I/O labelled transition systems (i.e. trees) with a strict alternation between an input action and the set of output actions. After an output action we may find either a leaf or another input action. Leaves can be labelled either by *pass* or by *fail*. In addition to check the functional behavior of the IUT, test have also to detect whether wrong timed behaviors appear. Thus, tests have to include capabilities to deal with the two ways of specifying time. On the one hand, we will include *random variables*. The idea is that we will record the time that the implementation takes to arrive to the leaves of the test labelled with *pass*. We will collect a sample of times for each test execution and we will *compare* this sample with the random variable associated to the leaf reached in the test. By comparison we mean that we will apply a contrast to decide, with a certain confidence, whether the sample could be generated by the corresponding random variable. On the second hand, tests will include *delays* before offering input actions. The idea is that delays in tests will induce timeouts in IUTs. Thus, we may indirectly check whether the timeouts imposed by the specification are reflected in the IUT by offering input actions after a specific delay.

Definition 8. A *test case* is a tuple $T = (S, I, O, \lambda, s_0, S_I, S_O, S_F, S_P, \zeta, D)$ where S is the set of states, I and O, with $I \cap O = \emptyset$ are the sets of input and output actions, respectively, $\lambda \subseteq S \times I \cup O \times S$ is the transition relation, $s_0 \in S$ is the initial state, and the sets $S_I, S_O, S_F, S_P \subseteq S$ are a partition of S. The transition relation and the sets of states fulfill the following conditions:

- S_I is the set of *input* states. We have that $s_0 \in S_I$. For all input state $s \in S_I$ there exists a unique outgoing transition $(s, a, s') \in \lambda$. For this transition we have that $a \in I$ and $s' \in S_O$.
- S_O is the set of *output* states. For all output state $s \in S_O$ we have that for all $o \in O$ there exists a unique state s' such that $(s, o, s') \in \lambda$. In this case, $s' \notin S_O$. Moreover, there do not exist $i \in I, s' \in S$ such that $(s, i, s') \in \lambda$.
- S_F and S_P are the sets of *fail* and *pass* states, respectively. We say that these states are *terminal*. Thus, for all state $s \in S_F \cup S_P$ we have that there do not exist $a \in I \cup O$ and $s' \in S$ such that $(s, a, s') \in \lambda$.

Finally, we have two timed functions. $\zeta : S_P \longrightarrow \bigcup_{j=1}^{\infty} \mathcal{V}^j$ is a function associating random variables, to compare with the time that the implementation took to perform the outputs, with passing states. $D : S_I \longrightarrow \texttt{Time}$ is a function associating delays with input states.

We say that a test case T is *valid* if the graph induced by T is a tree with root at the initial state s_0. We say that a set of tests $\mathcal{T}_{st} = \{T_1, \ldots, T_n\}$ is a *test suite*.

Let $\sigma = i_1/o_1, \ldots, i_r/o_r$. We write $T \stackrel{\sigma}{\Longrightarrow} s^T$ if $s^T \in S_F \cup S_P$ and there exist states $s_{12}, s_{21}, s_{22}, \ldots s_{r1}, s_{r2} \in S$ such that $\{(s_0, i_1, s_{12}), (s_{r2}, o_r, s^T)\} \subseteq \lambda$, for all $2 \leq j \leq r$ we have $(s_{j1}, i_j, s_{j2}) \in \lambda$, and for all $1 \leq j \leq r-1$ we have $(s_{j2}, o_j, s_{(j+1)1}) \in \lambda$.

Let T be a valid test, $\sigma = i_1/o_1, \ldots, i_r/o_r$, s^T be a state of T, and $\overline{t} = (t_1, \ldots, t_r) \in \texttt{Time}^r$. We write $T \stackrel{\sigma}{\Longrightarrow_{\overline{t}}} s^T$ if $T \stackrel{\sigma}{\Longrightarrow} s^T$, $t_1 = D(s_0)$, and for all $1 < j \leq r$ we have $t_j = D(s_{j1})$. □

Let us remark that $T \stackrel{\sigma}{\Longrightarrow} s^T$, and its variant $T \stackrel{\sigma}{\Longrightarrow_{\overline{t}}} s^T$, imply that s^T is a terminal state. Next we define the application of a test suite to an implementation. We say that the test suite $\mathcal{T}_{st}$ is *passed* if for all test the terminal states reached by the composition of implementation and test are *pass* states. Besides, we give different timing conditions in a similar way to what we did for implementation relations.

Definition 9. Let I be $\texttt{SFSM}$ and $T = (S_t, I, O, \delta_T, s_0, S_I, S_O, S_F, S_P, \zeta, D)$ be a valid test, $\sigma = i_1/o_1, \ldots, i_r/o_r$, s^T be a state of T, $\overline{t} = (t_1, \ldots, t_r)$, and $\bar{t}_o = (t_{o1}, \ldots, t_{or})$. We write $I \,\|\, T \stackrel{\sigma}{\Longrightarrow_{\bar{t}}} s^T$ if $T \stackrel{\sigma}{\Longrightarrow_{\bar{t}}} s^T$ and $(\sigma, \bar{t}) \in \texttt{InsFEvol}(I)$. We write $I \parallel T \stackrel{\sigma}{\Longrightarrow_{\overline{t}, \bar{t}_o}} s^T$ if $I \parallel T \stackrel{\sigma}{\Longrightarrow_{\overline{t}}} s^T$ and $(\sigma, \bar{t}, \bar{t}_o)$ is a observed timed execution of I. In this case we say that $(\sigma, \bar{t}, \bar{t}_o)$ is a *test execution* of I and T.

We say that I *passes* the test suite $\mathcal{T}_{st}$, denoted by $\texttt{pass}(I, \mathcal{T}_{st})$, if for all test $T \in \mathcal{T}_{st}$ there do not exist $(\sigma, \overline{t}) \in \texttt{InsFEvol}(I)$, $s^T \in S$ such that $I \parallel T \stackrel{\sigma}{\Longrightarrow_{\overline{t}}} s^T$ and $s^T \in S_F$. □

Let us remark that since we are assuming that implementations are input-enabled, the testing process will conclude only when the test reaches either a fail or a pass state.

In addition to this notion of passing tests, we will have different time conditions. We apply the time conditions to the set of *observed timed executions*, not to stochastic evolutions of the implementations, due to the fact that stochastic evolutions do not have a single time value that we can directly compare with the time stamp attached to the pass state. In fact, we need a set of test executions associated to each evolution to evaluate if they match the distribution function associated to the random variable indicated by the corresponding state of the test. In order to increase the degree of reliability, we will not take the classical approach where passing a test suite is defined according only to the results for each test. In our approach, we will put together all the observations, for each test, so that we have more samples for each evolution. In particular, some observations will be used several times. In other words, an observation from a given test may be used to check the validity of another test sharing the same observed sequence.

Definition 10. Let I be a SFSM and $\mathcal{T}_{st} = \{T_1, \ldots, T_n\}$ be a test suite. Let $H_1, \ldots, H_n$ be multisets of test executions of I and $T_1, \ldots, T_n$, respectively. Let $H = \biguplus_{i=1}^{n} H_i$, $\Phi = \{(\sigma, \bar{t}) \mid \exists\, \bar{t}_o : (\sigma, \bar{t}, \bar{t}_o) \in H\}$, $0 \leq \alpha \leq 1$ and let us consider $\mathtt{Sampling}_{(H,\Phi)}$ and $\mathtt{Sampling}^k_{(H,\Phi)}$, for all $1 \leq k \leq max\{|\sigma| \mid (\sigma, \bar{t}) \in \Phi\}$.

Let $e = (\sigma, \overline{t}) \in \Phi$. We define the set $\mathtt{Test}(e, \mathcal{T}_{st}) = \{T \mid T \in \mathcal{T}_{st} \ \wedge\ I \parallel T \stackrel{\sigma}{\Longrightarrow}_{\overline{t}} s^T\}$.

We say that the implementation I *weakly* $(\alpha, H)-$*passes* the test suite $\mathcal{T}_{st}$ if $\mathtt{pass}(I, \mathcal{T}_{st})$ and for all $e = (\sigma, \bar{t}) \in \Phi$ we have that for all $T \in \mathtt{Test}(e, \mathcal{T}_{st})$ such that $I \parallel T \stackrel{\sigma}{\Longrightarrow}_{\overline{t}} s^T$ it holds $\gamma(\sum \zeta(s^T), \mathtt{Sampling}_{(H,\Phi)}(\sigma, \bar{t})) > \alpha$.

We say that the implementation I *strongly* $(\alpha, H)-$*passes* the test suite $\mathcal{T}_{st}$ if $\mathtt{pass}(I, \mathcal{T}_{st})$ and for all $e = (\sigma, \bar{t}) \in \Phi$ we have that for all $T \in \mathtt{Test}(e, \mathcal{T}_{st})$ such that $I \parallel T \stackrel{\sigma}{\Longrightarrow}_{\overline{t}} s^T$ it holds $\forall\ 1 \leq j \leq |\sigma| : \gamma(\pi_j(\zeta(s^T)), \mathtt{Sampling}^j_{(H,\Phi)}(\sigma, \bar{t})) > \alpha$. □

Let us note that an observed timed execution does not return the random variable associated with performing the evolution (that is, the addition of all the random variables corresponding to each transition of the implementation) but the time that it took to perform the evolution. Intuitively, an implementation passes a test if there does not exist an evolution leading to a fail state. Once we know that the functional behavior of the implementation is correct with respect to the test, we need to check time conditions. The set H corresponds to the observations of the (several) applications of the tests belonging to the test suite $\mathcal{T}_{st}$ to I. Thus, we have to decide whether, for each evolution e, the observed time values (that is, $\mathtt{Sampling}_{(H,\Phi)}(e)$) *match* the definition of the random variables appearing in the successful state of the tests corresponding to the execution of that evolution (that is, $\zeta(s^T)$). As we commented previously, we assume a function γ, formally defined in the appendix of [10], that can perform this hypothesis contrast.

5 Test Derivation: Soundness and Completeness

In this section we present an algorithm to derive test cases from specifications and we show that the derived test suites are sound and complete with respect to the two implementation relations presented in Section 3. As usual, the idea underlying our algorithm consists in traversing the specification in order to get all the possible traces in an appropriate way. First, we introduce some additional notation.

Definition 11. Let $M = (S, I, O, \delta, TO, s_{in})$ be a SFSM. We consider the following sets:

$$\texttt{out}(s,i) = \{o \mid \exists\, s', \xi : (s,i,o,s',\xi) \in \delta\}$$

$$\texttt{afterTO}(s,t) = \begin{cases} s & \text{if } \pi_2(TO(s)) > t \\ \texttt{afterTO}(\pi_1(TO(s)), t - \pi_2(TO(s))) & \text{otherwise} \end{cases}$$

$$\texttt{after}(s,i,o,\bar{\xi}) = \begin{cases} (s', \bar{\xi}') & \text{if } \exists\, \xi : (s,i,o,s',\xi) \in \delta \\ \texttt{error} & \text{otherwise} \end{cases}$$

where if $\bar{\xi} = (\xi_1, \ldots, \xi_n)$ then $\bar{\xi}' = (\xi_1, \ldots, \xi_n, \xi)$ □

The function $\texttt{out}(s,i)$ computes the set of output actions associated with those transitions that can be executed from s after receiving the input i. The next function, $\texttt{afterTO}(s,t)$ returns the state that would be reached by the system if we start in the state s and t time units elapsed without receiving an input. The last function, $\texttt{after}(s,i,o,\bar{\xi})$, computes the state reached from a state s after receiving the input i, producing the output o, supposing that $\bar{\xi}$ denotes the random variables associated to the transitions previously performed. In addition, it returns the new tuple of random variables associated to the transitions performed since the system started its performance. Let us also remark that due to the assumption that SFSMs are observable we have that $\texttt{after}(s,i,o,\bar{\xi})$ is uniquely determined. Besides, we will apply this function only when the side condition holds, that is, we will never receive error as result of applying after.

The algorithm to derive tests from a specification is given in Figure 2. It is a non-deterministic algorithm that returns a single test. By considering the possible available choices in the algorithm we extract a full test suite from the specification (this set will be infinite in general). For a given specification M, we denote this set of tests by $\texttt{tests}(M)$. Next we explain how the algorithm works. A set of *pending situations* S_{aux} keeps those triplets denoting the possible states and the tuple of random variables that could appear in a state of the test whose definition, that is, its outgoing transitions, has not been completed yet. A triplet $(s^M, \bar{\xi}, s^T) \in S_{aux}$ indicates that we did not complete the state s^T of the test, the tuple of random variables $\bar{\xi}$ associated to the transitions of the specification that have been traversed from the initial state, and the current state in the transversal of the specification is s^M.

Input: A specification $M = (S, I, O, \delta, TO, Tr, s_{in})$.
Output: A test case $T = (S', I, O \cup \{\texttt{null}\}, \lambda, s_0, S_I, S_O, S_F, S_P, \zeta, D)$.

Initialization:

- $S' := \{s_0\}, \delta := S_I := S_O := S_F := S_P := \zeta := D := \emptyset$.
- $S_{aux} := \{(s_{in}, null, s_0)\}$.

Inductive Cases: Choose one of the following two options until $S_{aux} = \emptyset$.

1. If $(s^M, \bar{\xi}, s^T) \in S_{aux}$ then perform the following steps:
   (a) $S_{aux} := S_{aux} - \{(s^M, \bar{\xi}, s^T)\}$.
   (b) $S_P := S_P \cup \{s^T\}$; $\zeta(s^T) := \bar{\xi}$.
2. If $S_{aux} = \{(s^M, \bar{\xi}, s^T)\}$ and $\exists\, t_d \in \texttt{Time}$, $i \in I$ such that $\texttt{out}(\texttt{afterTO}(s^M, t_d), i) \neq \emptyset$ then perform:
   (a) Choose $t_d \in \texttt{Time}$ and $i \in I$ fulfilling the previous conditions.
   (b) $s^M = \texttt{afterTO}(s^M, t_d)$; $S_{aux} := \emptyset$.
   (c) Consider a fresh state $s' \notin S'$ and let $S' := S' \cup \{s'\}$.
   (d) $S_I := S_I \cup \{s^T\}$; $S_O := S_O \cup \{s'\}$; $\lambda := \lambda \cup \{(s^T, i, s')\}$.
   (e) $D(s^T) := t_d$.
   (f) For all $o \notin \texttt{out}(s^M, i)$ do `{null is in this case}`
      - Consider a fresh state $s'' \notin S'$ and let $S' := S' \cup \{s''\}$.
      - $S_F := S_F \cup \{s''\}$; $\lambda := \lambda \cup \{(s', o, s'')\}$.
   (g) For all $o \in \texttt{out}(s^M, i)$ do
      - Consider a fresh state $s'' \notin S'$ and let $S' := S' \cup \{s''\}$.
      - $\lambda := \lambda \cup \{(s', o, s'')\}$.
      - $(s_1^M, \bar{\xi}') := \texttt{after}(s^M, i, o, \bar{\xi})$.
      - $S_{aux} := S_{aux} \cup \{(s_1^M, \bar{\xi}', s'')\}$.

Fig. 2. Derivation of test cases from a specification

Let us consider the steps of the algorithm. The set S_{aux} initially contains a tuple with the initial states (of both the specification and the test) and the initial tuple of random variables (that is, empty tuple of random variables). For each tuple belonging to S_{aux} we may choose one possibility between two choices. It is important to remark that the second choice can be taken only when the set S_{aux} becomes singleton. So, our derived tests correspond to valid tests as given in Definition 8. The first possibility simply indicates that the state of the test becomes a passing state. The second possibility takes an input and generates a transition in the test labelled by this input. At this step, we choose a delay for the next input state. We select a time value and replace the states of the pending situation by the situation that can be reached if we apply as delay for accepting a new input, the time value selected. This is because, during the delay, the timeout transition associated to the state s_M can be triggered, so a change of state will be prompted by this fact. That fact allow us to consider sequences of

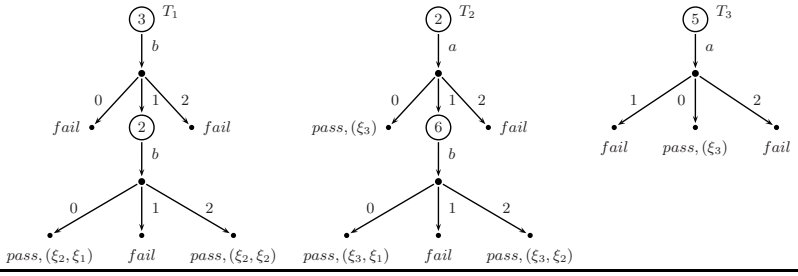


Fig. 3. Examples of Tests

timeout transitions, that is, traces where those transitions are triggered because no input action is received by the system.

Then, the whole sets of outputs is considered. If the output is not expected by the implementation (step 2.(f) of the algorithm) then a transition leading to a failing state is created. This could be simulated by a single branch in the test, labelled by `else`, leading to a failing state (in the algorithm we suppose that *all* the possible outputs appear in the test). For the expected outputs (step 2.(g) of the algorithm) we create a transition with the corresponding output action and add the appropriate tuple to the set S_{aux}.

Finally, let us remark that finite test cases are constructed simply by considering a step where the second inductive case is not applied. Finally, let us comment on the *finiteness* of our algorithm. If we do not impose any restriction on the implementation (e.g., a bound on the number of states) we cannot determine some important information such as the maximal length of the traces that the implementation can perform. In other words, we would need a *coverage criterium* to generate a finite test suite. Since we do not assume, by default, any criteria, all we can do is to say that this is the, in general, infinite test suite that would allow to prove completeness. Obviously, one can impose restrictions such as "generate n tests" or "generate all the tests with m inputs" and *completeness* will be obtained up to that coverage criterium.

Example 3. In Figure 3 we present some examples of test cases. These tests are derived from the specification presented in Figure 1. In the test T_1 we consider a delay of 3 time units in the step 2.(a) of the algorithm as well as the input b. A transition labelled by this input is generated in the test. Next, all outputs are considered. Due to the fact that the specification only accepts the output 1, two transitions leading to a fail state are created for the outputs 0 and 2 respectively (step 2.(f) of the algorithm). Moreover, we create a transition for the output 1 (step 2.(g) of the algorithm). After this, we select again the input b and establish a delay of 2 time units for this input. The corresponding transitions are created in the test. Finally, we apply the step 1 of the algorithm in order to conclude the generation of this test. The pass states contain some random

variables, extracted from the specification, that will be used to compare the time values that the implementation takes to perform outputs with the ones that are presented in the specification. For instance, the tuple (ξ_2, ξ_1) that appears in the pass state of the left branch of T_1 is extracted from the transitions $s_1 \xrightarrow{b/1}_{\xi_2} s_4$ and $s_4 \xrightarrow{b/0}_{\xi_1} s_1$. Regarding the test T_2, let us note that the number of random variables associated with the pass states varies depending on the level in which it is derived. That is because we generate a tuple of random variables that presents so many elements as pairs of input/outputs have been transversed in the specification. The tests T_2 and T_3 consider the same input in the first transition a. The difference lies in the delays we consider for each of them, 2 and 5 time units, respectively. This fact makes that for the test T_2 the output 1 is accepted. However, in the test T_3 it leads to a fail state, because in this case the timeout associated to the initial state should be triggered after 4 time units and the machine would change its state to s_2, where the output 1 is not accepted for the input a. □

Finally, we present the result that relates, for a specification S and an implementation I, implementation relations and application of test suites.

Theorem 1. Let S, I be SFSMs. Let H be a multiset of test executions of I, $0 \leq \alpha \leq 1$, and $\Phi = \{(\sigma, \bar{t}) \mid \exists\, \bar{t}_o : (\sigma, \bar{t}, \bar{t}_o) \in H\} \cap \mathtt{InsFEvol}(S)$. We have that:

- $I\ \mathtt{confs}_w^{(\alpha,H)}\ S$ iff I *weakly* $(\alpha, H)-$*passes* $\mathtt{tests}(S)$.
- $I\ \mathtt{confs}_s^{(\alpha,H)}\ S$ iff I *strongly* $(\alpha, H)-$*passes* $\mathtt{tests}(S)$.

□

6 Concluding Remarks

This paper concludes the work initiated in [10]. There, we presented a new notion of finite state machine to specify, in an easy way, both the passing of time due to timeouts and the time due to the performance of actions. In addition, we presented several implementation relations based on the notion of conformance. These relations shared a common pattern: The implementation must conform to the specification regarding functional aspects. In this paper we introduce a notion of test, how to apply a test suite to an implementation, and what is the meaning of successfully passing a test suite. Even though implementation relations and passing of test suites are, apparently, unrelated concepts, we provide a link between them: We give an algorithm to derive test suites from specifications in such a way that a test suite is successfully passed iff the implementation conforms to the specification. This result, usually known as *soundness* and *completeness*, allows a user that in order to check the correctness of an implementation, it is the same to consider an implementation relation or to apply a derived test suite.

References

1. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Brandán, L., Brinksma, E.: Testing real-time multi input-output systems. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 264–279. Springer, Heidelberg (2005)
3. Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 187–195. Springer, Heidelberg (2001)
4. Clarke, D., Lee, I.: Automatic generation of tests for timing constraints from requirements. In: 3rd Workshop on Object-Oriented Real-Time Dependable Systems, WORDS'97, pp. 199–206. IEEE Computer Society Press, Los Alamitos (1997)
5. En-Nouaary, A., Dssouli, R., Khendek, F.: Timed Wp-method: Testing real time systems. IEEE Transactions on Software Engineering 28(11), 1024–1039 (2002)
6. Krichen, M., Tripakis, S.: An expressive and implementable formal framework for testing real-time systems. In: Khendek, F., Dssouli, R. (eds.) TestCom 2005. LNCS, vol. 3502, pp. 209–225. Springer, Heidelberg (2005)
7. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using Uppaal. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 79–94. Springer, Heidelberg (2005)
8. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines: A survey. Proceedings of the IEEE 84(8), 1090–1123 (1996)
9. Mandrioli, D., Morasca, S., Morzenti, A.: Generating test cases for real time systems from logic specifications. ACM Transactions on Computer Systems 13(4), 356–398 (1995)
10. Merayo, M.G., Núñez, M., Rodríguez, I.: Implementation relations for stochastic finite state machines. In: Horváth, A., Telek, M. (eds.) EPEW 2006. LNCS, vol. 4054, pp. 123–137. Springer, Heidelberg (2006)
11. Núñez, M., Rodríguez, I.: Encoding PAMR into (timed) EFSMs. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 1–16. Springer, Heidelberg (2002)
12. Núñez, M., Rodríguez, I.: Towards testing stochastic timed systems. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 335–350. Springer, Heidelberg (2003)
13. Núñez, M., Rodríguez, I.: Conformance testing relations for timed systems. In: Grieskamp, W., Weise, C. (eds.) FATES 2005. LNCS, vol. 3997, pp. 103–117. Springer, Heidelberg (2006)
14. Petrenko, A.: Fault model-driven test derivation from finite state models: Annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 196–205. Springer, Heidelberg (2001)
15. Rodríguez, I., Merayo, M.G., Núñez, M.: HOTL: Hypotheses and observations testing logic. In: Journal of Logic and Algebraic Programming (2007), http://dx.doi.org/10.1016/j.jlap.2007.03.002
16. Springintveld, J., Vaandrager, F., D'Argenio, P.R.: Testing timed automata. Theoretical Computer Science, 254(1-2):225–257, 2001. Previously appeared as Technical Report CTIT-97-17, University of Twente (1997)